Correction Mines-Ponts 2018

PSI 2022/2023

Partie I - stockage interne des données

☐ **Q1.** Une fréquence d'échantillonnage de 2 Hz signifie une mesure toutes les 0.5 s. En 20 minutes, il y a donc $20 \times 120 = 2400$ mesures. Chaque mesure contient 8 caractères, elle est donc codée sur 8 octets (car un caractère est codé sur 8 bits = 1 octet). Un enregistrement nécessite donc $8 \times 2400 = 19{,}2$ ko de mémoire.

☐ **Q2.** Il y a un enregistrement toutes les 30 minutes. En 15 jours, cela fait un total de $15 \times 24 \times 2 = 720$ enregistrements. On aura besoin en tout d'un peu plus de $720 \times 19,2.10^3 = 13,8$ Mo. Une carte mémoire de 1 Go est donc largement suffisante.

☐ **Q3.** Sachant que chaque mesure contient 8 caractères, en supprimant 1 caractère pour chaque mesure on supprime $1/8^e$ du fichier, on économise $1/8^e$ de la mémoire. On aura donc un gain en mémoire de $13{,}8/8 = 1{,}73$ Mo.

☐ **Q4.** On rappelle les fonctions à connaître concernant l'ouverture/fermeture des fichiers :

— `open ("nom_du_fichier",mode)` permet d'ouvrir le fichier dans une variable de type "file". `mode` est un caractère précisant si on souhaite ouvrir le fichier en lecture (`r`) seule ou en écriture (`w`).
— `readlines()` renvoie la ligne courante de la variable de fichier sous la forme d'une chaîne de caractères.
— `readlines()` renvoie toutes les lignes de la variable de fichier sous la forme d'une liste.

Je vous conseille de tester ces fonctions ou de revoir le TP6 sur la Dropbox.

```python
fichier = open("donnees.txt","r")
lignes_mesures = fichier.readlines()
liste_niveaux = []
for i in range(1,len(lignes_mesures)):
    liste_niveaux.append(float(lignes_mesures[i]))
fichier.close()
```

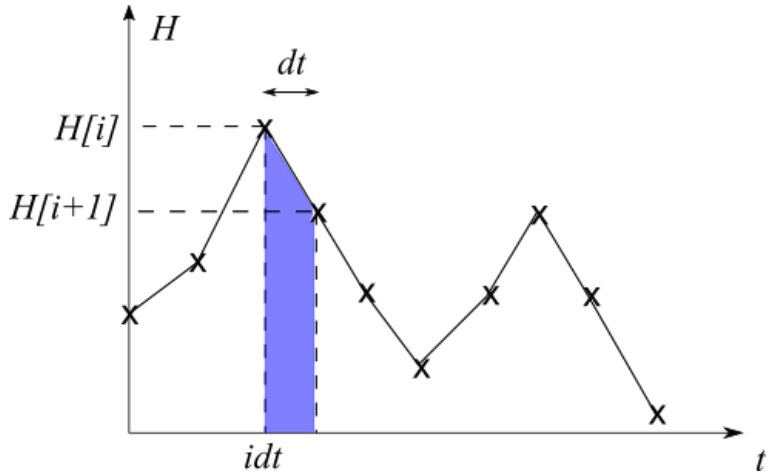
Partie II -Analyse "vague par vague"

☐ **Q5.** On lit approximativement $H_1 = 9$m, $H_2 = 9$m et $H_3 \approx 6$m. Par ailleurs $T_1 \approx 12$s et $T_2 \approx 13$s.

☐ **Q6.**

```python
def moyenne(liste_niveaux):
    moy = 0
    for x in liste_niveaux:
        moy = moy + x
    moy /= len(liste_niveaux)
    return moy
```

☐ **Q7.** Dans la méthode des trapèzes, l'aire sous une fonction H dans un intervalle $[a, b]$ est approximée par la somme des aires des trapèzes sous la courbe.



Ici, les mesures étant séparées par des intervalles de temps égaux valant $\Delta t = 0,5$s, on obtient :

$$\mathcal{A} = \Delta t \sum_{i=0}^{n-2} \frac{L[i] + L[i+1]}{2} = \Delta t \times \left(\frac{\mathtt{L[0]}}{2} + \mathtt{L[1]} + ... + \mathtt{L[n-2]} + \frac{\mathtt{L[n-1]}}{2} \right)$$

Le code s'écrira donc :

```python
def integrale_precise(liste_niveaux):
  n = len(liste_niveaux)
  aire = liste_niveaux[0]/2
  dt = 0.5
  for i in range(n-1):
    aire += liste_niveaux[i]
    aire += liste_niveaux[n-1]
    aire *= dt
  return aire
```

La fonction suivante s'en déduit :

```python
def moyenne_precise(liste_niveaux):
  dt = 0.5
  t_tot = len(liste_niveaux) * dt
  return integrale_precise(liste_niveaux)/t_tot
```

☐ **Q8.**

```python
def ind_premier_pzd(liste_niveaux)
   moy = moyenne_precise(liste_niveaux)
   n = len(liste_niveaux)
   for i in range(n-1):
      if liste_niveaux[i] > moy and liste_niveaux{i+1] < moy
         return i
   return -1
```

☐ **Q9.** Il suffit de parcourir la liste à l'envers afin que la fonction soit bien de complexité $\mathcal{O}(1)$ dans le meilleur des cas (cas où l'indice recherché correspond à l'avant-dernier élément de la liste). La façon la plus simple est d'utiliser une boucle `while` :

```
def ind_dernier_pzd(moyenne,liste_niveaux)
  n = len(liste_niveaux)-1
  while not(liste_niveaux[n] < moyenne and liste_niveaux[n-1] > moyenne)
   and n >0:
    n -= 1
  if n == 0:
    return -2
  else:
    return n
```

Attention piège vicieux : il ne faut pas intégrer le calcul de la moyenne dans la fonction car ce calcul est en $\mathcal{O}(n)$. Il faut donc appeler la fonction `moyenne` en dehors de la fonction et rajouter la moyenne en argument.

☐ **Q10.**

```
def construction_successeurs(liste_niveaux):
  n=len(liste_niveaux)
  successeurs=[]
  m=moyenne(liste_niveaux)
  for i in range(n-1):
          if liste_niveaux[i] > moy and liste_niveaux[i+1] < moy
                  successeurs.append(i+1)
  return successeurs
```

☐ **Q11.**

```
def decompose_vagues(liste_niveaux):
  successeurs = construction_successeurs (liste_niveaux)
  liste_vagues = []
  for i in range(len(successeurs)-1):
    liste_vagues.append(liste_niveaux[successeurs[i]:successeurs[i+1])
  return liste_vagues
```

☐ **Q12.** Il faut faire attention au fait que la hauteur d'une vague i est définie comme la différence du `max` de la vague $i-1$ et du `min` de la vague i (le sujet n'est pas très logique...). Par ailleurs, il faut traiter le premier cas de manière séparé puisque le premier maximum ne fait parti d'aucune vague :

```
def proprietes(liste_niveaux):
  vagues = decompose_vagues(liste_niveaux)
  dt = 0.5 # Intervalle de temps entre deux points
  Prop = [] # La future liste des propriétés que l'on veut obtenir

  # On traite à part le premier cas
  i0 = ind_premier_pzd(liste_niveaux)
  H1 = max(liste_niveaux[:i0]) - min(vagues[0])
  T1 = len(vagues[0])*dt
  Prop.append([H1,T1])

  # Après, on peut prendre les vagues les unes après les autres.
  for i in range(len(vagues)-1):
    Hi = max(vagues[i]) - min(vagues[i+1])
    Ti = len(vagues[i+1])*dt
    Prop.append([Hi,Ti])
  return Prop
```

Partie III - Contrôle des données

☐ **Q13.** On utilise bien entendu la fonction `proprietes` précédente et on recherche le maximum de la liste de liste par rapport au premier élément :

```
def H_max(liste_niveaux):
  Prop = propriete(liste_niveaux)
  Hmax = 0
  for Pi in Prop:
    if Pi[0] > Hmax:
      Hmax = Pi[0]
  return Hmax
```

☐ **Q14.** Au premier appel de la fonction de tri, on doit regarder la liste en totalité. Par conséquent, l'indice `g` doit valoir initialement 0 alors que l'indice `d` doit valoir `len(liste)-1` (attention, il ne peut valoir `len(liste)` car la fonction appelle `liste[d][0]`). Pour le choix du pivot, on peut prendre n'importe quelle hauteur de vague, par exemple `pivot = liste[d][0]`, mais on aurait tout aussi bien pu prendre `pivot = liste[g][0]` ou bien `pivot = liste[(g+d)//2][0]`.

☐ **Q15.** Il suffit de rajouter en toute première ligne de la fonction `triRapide` la vérification sur la taille `d-g` restant à vérifier :

```
def triRapide(liste,g,d):
  if d - g < 15:
    triInsertion(liste,g,d)
  else:
```

On aurait pu, de manière équivalente, écrire deux conditions de ce type aux lignes 17 et 19.

☐ **Q16.** C'est un tri par insertion classique, exactement comme lorsque vous écartez vos cartes pour laisser passer la carte que vous voulez rajouter dans votre jeu en partant du côté droit. Ici la carte à insérer est l'élément `liste[i]`. Tant que cet élément est plus petit que l'élément qui est directement à sa gauche, on décale ce dernier d'un cran vers la droite. La comparaison se fait sur le premier élément des listes de listes.

```
def  triInsertion (liste, g, d):
  for i in range(g+1,d+1):
  j=i-1
  tmp = liste[i]
  while tmp[0] < liste[j][0] and j >=g:
    liste[j+1] = liste[j]
    j -= 1
  liste[j+1]=tmp
  return liste
```

☐ **Q17.** Dans la fonction `skewness` proposée, on peut remarquer à la ligne 6 que la moyenne est recalculée pour chacune des hauteurs de la liste alors que sa valeur est fixe et pourrait n'être calculée qu'une unique fois. Comme un calcul de moyenne est forcément linéaire en la taille de la liste initiale, le faire pour chacune des n valeurs de hauteurs produit une fonction globalement quadratique en n, ce qui n'est guère optimal ici. Il suffit de rajouter le calcul `moy = moyenne(liste_hauteurs)` entre les lignes 4 et 5 (juste avant la boucle `for`)

et utiliser cette valeur calculée dans la ligne 6 pour s'affranchir de ce souci.

☐ **Q18.** Les calculs de S et K font chacun intervenir une unique boucle sur les valeurs de la liste (pour calculer la somme). Il ne devrait donc pas y avoir de différence de type de la complexité si les deux fonctions procédant à cette évaluation ont été programmées de la même façon.

Partie IV - Base de données relationnelles

☐ **Q19.** (a) La première requête est une simple sélection sur la localisation :

```
SELECT idBouee,nomSite FROM Bouee WHERE localisation = 'Mediterranee'
```

(b) Pour la deuxième, il s'agit de vérifier que l'identifiant de la bouée n'est pas dans la table `Tempete`.

— on peut utiliser le mot clé `IN` :

```
SELECT idBouee  FROM Bouee
WHERE idBouee NOT IN(SELECT idBouee FROM Tempete)
```

— On peut aussi procéder à l'aide d'un `COUNT(*)` en utilisant un alias comme suit :

```
SELECT idBouee AS id FROM Bouee
WHERE (SELECT COUNT(*) FROM Tempete WHERE idBouee=id) = 0
```

La requête sélectionne tous les identifiants de `Bouee` dont le nombre d'apparition dans `Tempete` est égal à 0.

(c) Il faut effectuer une jointure et calculer le maximum pour chaque groupe de vagues associé à un lieu donné :

```
SELECT nomSite, MAX(Hmax)
FROM Bouee JOIN Tempete ON Bouee.idBouee=Tempete.idBouee
GROUP BY nomSite
```

Partie V - Analyse spectrale

☐ **Q20.** Il s'agit d'un algorithme de type "diviser pour régner". La complexité en calcul vérifie la relation de récurrence :

$$C(N) = 2C\left(\frac{N}{2}\right)$$

relation qu'on trouve aussi pour le tri fusion. On peut donc supposer que la complexité sera en moyenne de $\boxed{\mathcal{O}(N \log N)}$.

☐ **Q21.** Pour écire un code récursif, on se demandera toujours :

— quelle est la condition d'arrêt ?

— quel paramètre faut-il passer en argument de la fonction et comment varie-t-il au cours de chaque appel récursif ?

Ici, l'argument est donné par le sujet, il s'agit de la liste `x`. A chaque étape, on sépare les éléments d'indice pair et les éléments d'indice impair. Le code s'arrête lorsque la taille de la liste donnée en argument vaut 1.

En utilisant le *slicing* de listes :

```python
def TFD(x):
  n = len(x)
  if n == 1: #i = 0 dans la somme
    return x[0]
  else:
    pairs = x[0:n:2] #On part de 0 et on va de 2 en 2 jusqu'à n.
    impairs = x[1:n:2] # On part de 1 et on va de 2 en 2 jusqu'à n.
    P = TFD(pairs)
    I = TFD(impairs)
    omega = exp(-2*np.pi*1j/n)
    X = []
    for k in range(N//2)
      X.append(P[k]+omega**k*I[k])
    for k in range(N//2)
      X.append(P[k]-omega**k*I[k])
    return X
```